Coding Malware in Fancy Programming Languages for Fun and Profit

Apostolopoulos, Theodoros; Koutsokostas, Vasilios; Totosis, Nikolaos; Patsakis, Constantinos; Smaragdakis, Georgios

DOI
10.1145/3714393.3726506

Publication date
2025

Document Version
Final published version

Published in
CODASPY 2025 - Proceedings of the 15th ACM Conference on Data and Application Security and Privacy

Coding Malware in Fancy Programming Languages for Fun and Profit

Theodoros Apostolopoulos
University of Piraeus
Piraeus, Greece

Vasilios Koutsokostas
University of Piraeus
Piraeus, Greece

Nikolaos Totosis
University of Piraeus
Piraeus, Greece

Constantinos Patsakis
University of Piraeus & Athena Research Center
Piraeus, Greece

Georgios Smaragdakis
Delft University of Technology
Delft, The Netherlands

Abstract

The continuous increase in malware samples, both in sophistication and number, presents many challenges for organizations and analysts, who must cope with thousands of new heterogeneous samples daily. This requires robust methods to quickly determine whether a file is malicious. Due to its speed and efficiency, static analysis is the first line of defense.

In this work, we illustrate how the practical state-of-the-art methods used by antivirus solutions may fail to detect evident malware traces. The reason is that they highly depend on very strict signatures where minor deviations prevent them from detecting shellcodes that otherwise would immediately be flagged as malicious. Thus, our findings illustrate that malware authors may drastically decrease the detections by converting the code base to less-used programming languages. To this end, we study the features that such programming languages introduce in executables and the practical issues that arise for practitioners to detect malicious activity.

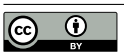
CCS Concepts

• **Security and privacy → Malware and its mitigation**; • **Software and its engineering → Compilers**; **General programming languages**.

Keywords

Malware, Evasion, Programming languages, Compilers

ACM Reference Format:
Theodoros Apostolopoulos, Vasilios Koutsokostas, Nikolaos Totosis, Constantinos Patsakis, and Georgios Smaragdakis. 2025. Coding Malware in Fancy Programming Languages for Fun and Profit. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY '25), June 4–6, 2025, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3714393.3726506



CODASPY '25, Pittsburgh, PA, USA

ACM ISBN 979-8-4007-1476-4/2025/06
https://doi.org/10.1145/3714393.3726506

1 Introduction

In the past decade, malware has undergone significant changes. The main drivers of these changes can be attributed to the vast digitization of products and services and the development of a payment system that allows anonymous transactions to bypass the protections of the traditional banking system. The former has boosted the number of possible victims and the potential impact of malware. Moreover, anonymous payment methods enable a wide array of illicit transactions to be performed, which, in the case of malware, is the apparent case of ransomware. Both the US Cybersecurity and Infrastructure Security Agency (CISA) [46, 47] and the European Union Agency for Cybersecurity (ENISA) [14] have recognized malware as the top cyber threat. Indeed, malware attacks impact our everyday lives by harvesting sensitive information, crippling critical services, and causing significant damage to individuals and corporations [46]. This has placed malware in a pivotal role in the crime ecosystem and created an individual ecosystem with independent roles operating in a business model called Malware-as-a-Service [36].

The security industry's response to the abovementioned threats is collecting and analyzing malware samples. At a rate of around 280,000 malware samples per day in 2024 [7], which is more or less similar to previous years, static analysis remains the most effective and profound remedy to detect malicious files quickly. In this arms race between malicious actors and defenders, the development of malware has evolved into an underground industry to bypass security controls by employing malware authors and monetizing the infected hosts. Of course, bypassing static analysis does not grant them a foothold to the targeted host. Nevertheless, it significantly raises their chances of achieving their goal, as they often need to bypass behavioral checks. Although endpoint detection and response systems usually apply such checks, and vendors often portray them as silver bullets, there are several ways to bypass them [17]. In this work, we limit our scope to static analysis.

Even though malware written in C continues to be the most prevalent (see our analysis in Section 3), malware operators, primarily known threat groups such as APT29 [28], increasingly include non-typical malware programming languages in their arsenal. For instance, APT29 recently used Python in their Masepie malware against Ukraine [9], while in their Zebrocy malware, they used a mixture of Delphi, Python, C#, and Go [42]. Likewise, Akira ransomware shifted from C++ to Rust [34], BlackByte ransomware

shifted from C# to Go [48], and Hive was ported to Rust [33]. According to the reports, the above changes exhibited increased resistance to reverse engineering and a low detection rate or misclassification.

On other occasions, C-language malware families are not recreated from scratch. Instead, malware authors write loaders, droppers, and wrappers in *"exotic"* languages. This provides them with several advantages, e.g., bypassing signature-based detection, so they can effectively place their payloads in harder-to-detect shells that are newly built. Thus, attackers continue to use the same initial penetration vector and a significant portion of their methods, suggesting that threat actors prefer to transfer the original malware code to different languages instead of modifying their tactics, techniques, and procedures (TTPs) to avoid detection. This approach allows them to maintain the effectiveness of their attacks while remaining under the radar of security systems. Since these languages may be less widely recognized or understood, they add an extra layer of obfuscation to malware, making it harder to detect and analyze. Furthermore, security analysts have reported increased difficulty in reverse engineering such malware samples due to reprogramming efforts [32]. Thus, combining different languages and obfuscation techniques complicates dissecting the malware's structure, functionality, and intent.

Our work explores the problem of detecting malware written in uncommon languages using a data-driven approach. Rather than merely reporting and examining this trend, we performed a targeted experiment by writing malicious samples in different programming languages and compilers and drilling down to the distinctive characteristics. This analysis practically shows the unique features that adversaries gain and highlights the emerging issues for malware detection and analysis.

The above leads to the formulation of some interesting research questions that have not been systematically studied in the academic literature, and we try to answer them in this work:

RQ1: How does the programming language and compiler choice impact the malware detection rate?
RQ2: What is the root cause of this disparity?
RQ3: What are the benefits of an attacker shifting the codebase to less common pairs of programming language and compiler beyond the detection rate by static analysis?

The remainder of this article is structured as follows. In the following section, we provide an overview of the related work. Next, we detail our motivation, formalize our research questions, and define our methodology. Then, in Section 4, we present our experiments and report our findings. The latter led us to examine the intrinsic differences when reverse engineering a binary in a non-standard programming language. We discuss our findings in Section 5, and finally, the work concludes, summarizing our findings and contributions and proposing ideas for future research.

2 Related work

Previous but sparse research has demonstrated how the runtime mechanics of programming languages or compiler characteristics have been exploited to evade static analysis and hinder reverse engineering. For example, Wang et al. [51] introduced the concept of translingual obfuscation, which leverages the unique features of logic programming languages like Prolog to obscure both data layout and control flow of C programs, complicating reverse engineering efforts. Their tool, BABEL, translates C functions into Prolog predicates, leveraging Prolog's unification and backtracking to create obfuscation layers resistant to static and dynamic analysis. In [24], binary obfuscation has been achieved using Continuation-Passing Style (CPS) transformation, which shares similar ideas with the intermediate code produced by functional language compilers such as Haskell. CPS transformation converts control flow into continuations, which severely fragment control flow graphs (CFGs), making static analysis and reverse engineering significantly more complex. Similarly, Lambda Obfuscation proposed by Lan et al. [21] uses lambda calculus to obfuscate program control flow and conceal sensitive branch conditions. Replacing conditional instructions with lambda expressions prevents adversaries from leveraging symbolic execution tools to recover a program's internal logic, hindering reverse engineering efforts. In a similar train of thought, Wang et al. [52] try to obfuscate the program execution flow by simulating Turing machines under branch conditions. Pawlowski et al. [38] obfuscate the control flow by making the execution probabilistic so that the execution traces differ per execution, even on the same input, confusing this way the analyst.

Romano et al. [40] partially translated parts of JavaScript to WebAssembly to make their malware evasive. Koutsokostas and Patsakis [18] introduced another evasion method focusing on Python malware packaged using PyInstaller. They illustrated how AV systems inherently struggle to detect Python bytecode, allowing malware authors to evade static analysis by exploiting the packaging noise PyInstaller adds to executables. In another study [8], Casolare et al. explored malware models that exploit the dynamic features of languages like Java. By leveraging Java's dynamic compilation, reflection, and class loading mechanisms, they managed to escape device antimalware software and signature detectors. Finally, in [39], a comparative analysis of real-world malware written in C and Rust is presented, and a dedicated framework that can easily analyze Rust malware is proposed, stressing the lack of academic research on Rust malware.

For years, ransomware groups have been switching to newer, unconventional languages to make reverse engineering and detection more difficult. Moreover, various threat actors have used this approach, employing a wide range of programming languages and techniques to obfuscate their malicious code. In [26], Visual Basic 6 binaries were characterized as the "*most hated binaries*" among security researchers due to the complexity of reverse engineering the code to analyze malware as the tools to dissect such binaries were scarce at that time. Visual Basic 6, despite being an older language, introduced unique challenges in dissecting the malware's structure and functionality, hindering the efforts of researchers to understand and mitigate threats. The Flame malware, discovered in 2012, was dubbed "*the most complex malware ever found*" [44] at that time. It used the Lua scripting language, which was relatively uncommon in malware at that time. Incorporating Lua added a layer of obfuscation, making the malware more challenging to analyze and understand. The Duqu malware [10], also known as Stuxnet 2.0, was written primarily in C++. However, the unique assembly patterns observed in the compiled code initially led researchers

to believe that it was written in an unknown high-level object-oriented programming language. After seeking help from the community [19], Kaspersky Lab discovered that the unusual patterns were due to an old C++ compiler used in legacy IBM systems, which had generated the code. This revelation highlighted the challenges researchers face in understanding this complex malware. A virus called Grip contained a Brainfuck interpreter coded in Assembly to generate its keycodes. Brainfuck is an intentionally minimalistic and challenging-to-understand programming language. Another extreme example is presented in [27], where attackers leverage REBOL, a lightweight language, to establish a command-and-control environment, allowing them to execute commands remotely. By employing such obscure languages, threat actors further hindered the efforts of security researchers to analyze and reverse engineer their malicious code.

To obscure the first step of the infection process and avoid security measures that identify the most common types of malicious code, malware authors can simply "wrap" commodity malware in loaders and droppers written in exotic languages. Also, malware developers can completely rewrite the code of current malware to produce new varieties. For example, the RustyBuer [20] malware variant is a new form of the Buer malware loader. Both of these tactics are being abused by known threat actors. The Sednit group – also known as APT28 [9], Fancy Bear, Sofacy, and STRONTIUM, are among the groups that have adopted a multi-language kill chain with uncommon languages in its development process on several occasions. For instance, APT28 developed the Zebrocy backdoor in Go and then rewrote its downloader in Nim in 2019 after it was initially created in Delphi. APT28 continues to employ the same initial penetration vector and many of the same methods, implying that threat actors are more likely to change the original malware code to a different language rather than change their TTPs to avoid detection. Recently, the Tomiris APT group was spotted utilizing a polyglot arsenal of programming languages, including some uncommon or unconventional in malware development. This diversification approach appears to aim at equipping operators with “full-spectrum malware” capable of evading security products. In several observed instances, the actor persistently cycled through different language malware strains until one was successfully executed on the targeted machines [11].

Exotic programming languages provide extra levels of obfuscation that go beyond traditional security procedures. Also, these languages are less often used in malware, reverse engineers are less experienced with their implementation, and malware analysis tools and sandboxes have a hard time evaluating samples written in them. Malware rewrites disrupt the static signature produced for well-known malware families, and because there is no identifying signature, malware written in obscure languages frequently escapes unnoticed by antivirus software. Malware detection using signatures relies on the presence of specific static characteristics within a file that remain constant and do not require execution to be identified. When malware is built in a new language, static indicators (for example, YARA rules) become irrelevant or ineffective [13].

Malware samples written in uncommon languages can multiply the effort required for reverse engineering by a sufficient factor. Many of these languages, particularly functional ones (e.g., Haskell, Lisp), employ a vastly different execution model from traditional malware development languages like C. In addition, these languages often introduce a large number of functions to the executable as part of their standard environment, resulting in a bloated binary that makes even simple programs like "Hello world" contain thousands of functions (e.g., Dart and Go). Moreover, using unconventional programming languages also introduces additional challenges to analysts, such as indirect function calls, different evaluation models, error handling procedures, memory safety operations, and garbage collectors. They also contain unique data structures and calling conventions, "mangled" symbols, as well as unique stack and heap management systems. Specifically, functional languages are characterized by their use of immutable data structures, first-class functions, and lazy evaluation, which can result in code that is difficult to comprehend and reverse engineer. In addition, different compilation options or compiler versions can make analysis even more challenging by breaking usual reverse engineering patterns. Furthermore, each language’s macro and meta-programming capabilities can help to further obfuscate the binary and slow down the analysis. The combination of the aforementioned artifacts can easily confuse the malware analyst and state-of-the-art tools, leading them to an unproductive rabbit hole. According to [53], many security experts consider that alternative languages like Golang, Rust, and Delphi produce compiled programs that are significantly less straightforward to analyze compared to traditional C-based binaries. In fact, as stated, many consider using such languages to be a novel evasive technique and the lack of tools to deal with a rising problem, as existing tools may produce less accurate results. Also, recent actions like the project OxA11C [43] launched by Sentinel One and Intezer Team, which aims to develop a methodology to make reverse engineering of Rust malware more approachable, as well as develop new tools to help researchers showcase the extent of the problem in the malware analysis domain.

3 Motivation and Methodology

We used real-world public datasets to establish ground truth on the usage of various programming languages and compilers by malware authors. First, we used the latest export of the database of Malware Bazaar [1]. We limited the dataset to compiled files and, more precisely, Windows executables. At the time of writing, this export contains 399,043 Windows executable files, adhering to the portable executable format. In general, `EXE` files are consistently the file format with the highest number of submissions and detections [50]. We queried these files with their hashes in VirusTotal, collecting the detection rate and, where available, the programming language and compiler. As shown in Figure 1, there are trends in the usage of programming languages and compilers by malware authors. These trends do not follow the trends of the TIOBE index [45], e.g., Python and Java, the two programming languages most widely used, are not represented in the dataset. Nevertheless, the differences are stiffer. For instance, the deviations in the detection rate by programming language and compiler are more than apparent, Figure 2. One can clearly observe the deviation in the detection rate from the first submission to the latest one. Moreover, it is also apparent that this disparity is wider in the less-used programming languages and compilers. Notably, this disparity appears in both the original and latest detections. Even more alarmingly,

Figure 1: Distribution of the top 5 programming languages of samples per year.

in this segment, we observe the lowest detection rate in the latest detections, showing that even malware in well-known languages has a low detection rate when a less common compiler is used.

Close investigation shows that the programming language and compiler choice can significantly impact the detection rate; see Figure 2. While one would expect less used programming languages, e.g., Rust and Nim, to have worse detection rates because the sparsity of samples would not allow the creation of robust rules, the use of non-widely used compilers, e.g., Pelles C, Embarcadero Delphi, and Tiny C, has a more substantial impact on the detection rate.

Then, we moved on to a more specific dataset. More precisely, we used the dataset of González-Manzano et al. [15], which is focused on APTs, making our findings more focused. Limiting the dataset to PE executables (2,190 samples), one can clearly observe in Figure 3 that the malware authors have shifted from coding *only* in Microsoft C++ to using more languages and compilers. Indeed, as time goes by, APTs choose more diverse programming languages and compilers, e.g., Borland and Embarcadero Delphi, Borland and Microsoft C++, or Purebasic. Apparently, these trends are aligned with the findings of the larger Bazaar dataset.

To answer the research questions (see Section §1), we developed a specific methodology and performed some very targeted experiments. According to our methodology, first, we create a reference dataset with malicious binaries. The intention is to make it as heterogeneous as possible in terms of programming languages and compilers. Nevertheless, we deliberately add well-known payloads that are immediately flagged by antimalware engines and do not obfuscate the binaries. This way, we avoid possible biases that obfuscation methods can introduce. Then, we submit the binaries to VirusTotal to assess how detectable these samples are from commercial antimalware engines (RQ1). We analyze the binaries to determine their structural differences, use tools and custom scripts to quantify their differences at the binary level (RQ2), and examine the effort and drawbacks that a reverse engineer would have. The latter, along with the known advantages of some frameworks and programming languages, allow us to streamline the benefits of a

```
powershell -NoP -NonI -W Hidden -Exec Bypass -Command New-Object System.N
↪ et.Sockets.TCPClient($IP,$port);$stream=$client.GetStream();[byte[]]$
↪ bytes=0..65535|%{0};while(($i=$stream.Read($bytes,0,$bytes.Length))
↪ -ne 0){$data=(New-Object -TypeName
↪ System.Text.ASCIIEncoding).GetString($bytes,0,$i);$sendback=(iex $data
↪ 2>&1 | Out-String);$sendback2=$sendback + 'PS ' + (pwd).Path + '>
↪ ';$sendbyte=([text.encoding]::ASCII).GetBytes($sendback2);$stream.Wri
↪ te($sendbyte,0,$sendbyte.Length);$stream.Flush();};$client.Close()
```

Listing 1: Payload I - PowerShell reverse shell.

```
LPVOID addressPointer = VirtualAlloc(NULL, sizeof(shellcode), 0x3000, 0x40);
RtlMoveMemory(addressPointer, shellcode, sizeof(shellcode));
HANDLE handle = CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)addressPointer,
↪  NULL, 0, 0);
WaitForSingleObject(handle, -1);
```

Listing 2: Payload II - Vanilla shellcode execution in C.

malware author to shift her codebase into less-used programming languages or use less common compilers (RQ3).

4 Experiment

In this section, we empirically show that the challenges posed by uncommon programming languages create difficulties for malware analysts during reverse engineering and affect automated detection systems.

4.1 Setting Up the Experiment

We performed a scoped experiment to assess the robustness of static analysis methods against malware written in uncommon (less-used) programming languages and frameworks. To this end, we experimented with common and uncommon programming languages that can generate native standalone Windows PE files using either a compiler or a packager. We selected the languages so that they can interact with the Windows OS by exposing a system command or by interacting with the Windows API via built-in libraries or the Foreign Function Interface (FFI). From a macro perspective, interaction with the underlying OS is the bare bone of every malware, alongside networking and cryptographic functionality.

Additionally, we tried to cover as many programming paradigms as feasible, as long as the produced binaries were standalone and dependent dynamically only on the native Windows DLLs or the .NET Framework. We created executables containing very simple and well-known payloads to evaluate the detection rate of malware written in different programming languages. The payloads were chosen from lists of online reports containing the most critical MITRE techniques used by adversaries [29], particularly the T1059 Command and Scripting Interpreter [30] and the T1055 Process Injection [31]. The first class of executables issues a system command that calls Powershell to initiate a reverse shell via a well-known piped command (see Listing 1). In contrast, the second class executes shellcode using a standard sequence of Windows API functions (see Listing 2). The actual shellcode invokes an executable and acts as a loader. To keep the analysis consistent, we tried to construct homogeneous and simple samples in terms of translation from one language to another without employing any techniques of obfuscation, anti-analysis, or compiler optimization. The code and corresponding binaries are available on GitHub [6].

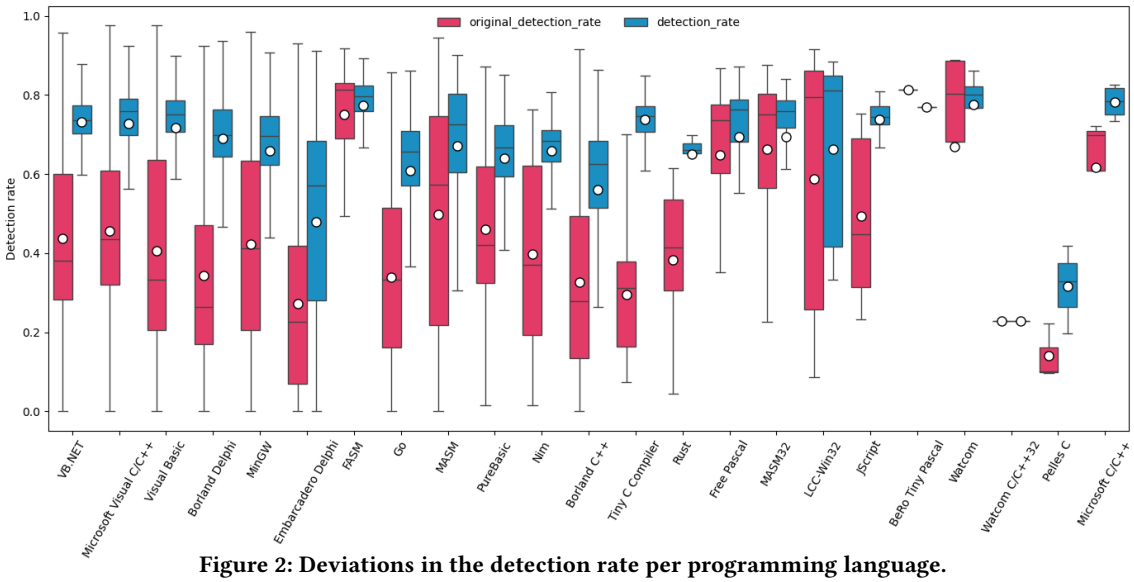


Figure 2: Deviations in the detection rate per programming language.

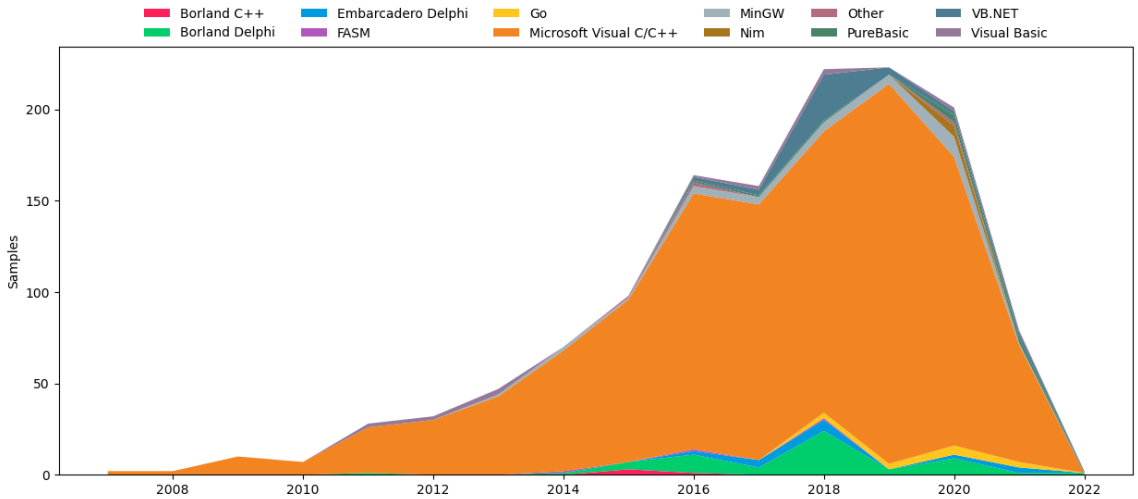


Figure 3: Distribution of top programming languages of APT samples per year from [15].

Then, we examine the binaries from two different standpoints. First, from the static analysis perspective using state-of-the-art antivirus engines and custom and open-source static analysis tools. Second, from the perspective of automated reverse engineering.

4.2 VirusTotal Results

In this part of the experiment, we used 39 programming languages and 50 different compilers or packagers to generate two samples for each possible payload, producing 100 unique samples. Then, we uploaded them to VirusTotal [49] and reported the detection results; see Table 1. It should be noted that, despite many of these samples being uploaded to VirusTotal for more than a year, a surprising number of them still remains undetected to date, even after rescans.

One can observe a great variance in the detection rate of the same payloads not only between different programming languages but also between compilers for the same programming language (see, for example, C/gcc vs C/dmc). Quite alarmingly, for the first payload, there are 13 samples with zero detections from AVs and 19 samples that reported very low detection rate (less than 5 AV engines), meaning that 32 of the 50 samples went undetected and had an overall detection rate of 6%. In the case of the shellcode payload, we had two samples with zero detections and 11 samples having very low detection rates with generic AV signatures like `Malicious(Moderate Confidence)`, `W64.AIDetectMalware`, `Unsafe`, `Python.Shell.6` (while not being a Python sample) and `Trojan.Malware.300983.susgen`, a notorious false positive detection (linked with many well-known benign binaries having the same false detection), while the overall detection rate was 21,7%. Our results clearly illustrate the inefficiency of static methods in

detecting the most simple malicious samples, even without any attempt to hide them. Figure 4 illustrates the variation of our samples in terms of the number of sections, threads, loaded DLLs, number of functions, and size. The figure clearly showcases that while all samples perform the same tasks and are all PE executables, structurally, they are radically different. The latter is also proven by the fact that even in terms of functions, there is even greater variation. More precisely, the number of functions ranges from 6 to 81,793; note that the figure is on a logarithmic scale to illustrate the results better.

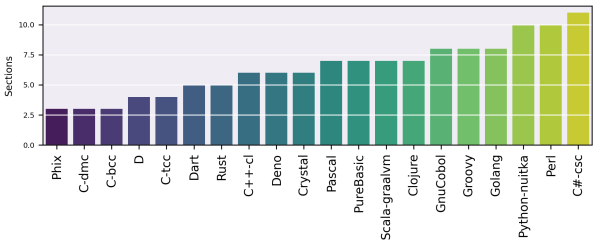
4.3 Open Source Static Analysis Tools

In this part of the experiment, we utilize *capa* [25], a robust, open-source capability-analysis tool developed by Mandiant widely used in cybersecurity environments, particularly within incident response teams, security operations centers (SOCs), and threat intelligence units. It can extract features from files, such as strings, disassembly, and control flow, and find combinations of features that are expressed in a common rule format. To gain ground truth, we first ran capa on the Assembly and C samples as they were the least bloated and straightforward and identified that a combination of the capa rules: `allocate or change RWX memory`, `create thread`, and `spawn thread to RWX shellcode` was able to correctly identify the shellcode execution basic block(s). Regarding the reverse Powershell payload, `execute command`,`create process on Windows`, or `accept command line arguments` were the rules that indicated system command invocation. Since some of these rules can be flagged even if they are harmless, as they may be just legal procedures inside the executables, we also verify them. For each sample, we check the reported address from capa with a debugger to determine whether it actually pointed to our malicious code, eliminating false positives. For example, the Haskell binary may report just `allocate or change RWX memory`, yet this was not for our malicious code. What is interesting is how well the results from VirusTotal correlate with the results from capa. Especially in the case of shellcode samples, we have an almost one-on-one correlation with the evasive samples, indicating that some unique structural characteristics let those samples go undetected.

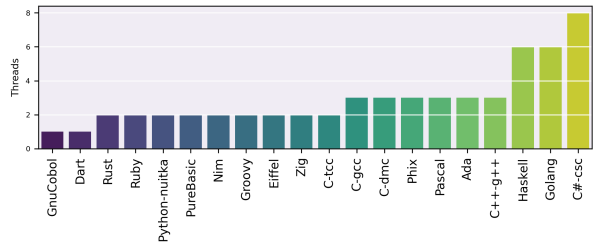
4.4 Shellcode Fragmentation

To assess how immune the binaries produced by different programming languages are to shellcode pattern matching, and since there is no intended obfuscation of the payloads, we conducted an experiment utilizing a custom-developed pattern-matching script to analyze the bytes from the raw binaries on disk. We allowed the matching operation to search for chunks of shellcode by fine-tuning two parameters for each binary, namely *Maximum Gap* and *Minimum Chunk Size*. For the former, we set the maximum allowed gap between matched shellcode bytes to 60, allowing flexibility for scattered patterns. For the latter, only matched sequences of at least 4 bytes were considered valid, reducing false positives from incidental matches of very small byte sequences and returning the sequences with the highest matching ratio.

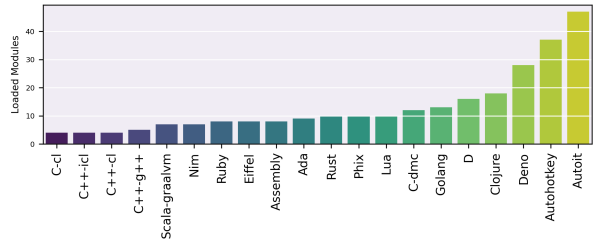
We also performed pattern matching in the reversed order of bytes to identify possible stack-based shellcodes (for example,



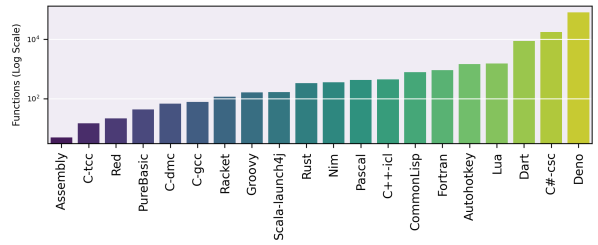
(a) Variation on the number of sections per payload/language.



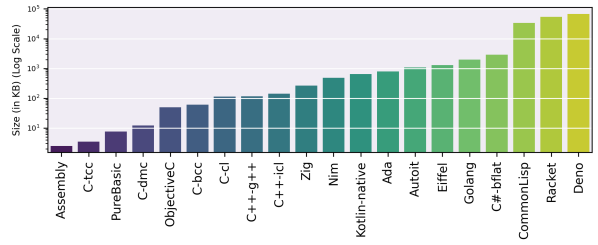
(b) Variation on the number of threads per payload/language.



(c) Variation on the number of loaded DLLs per language/language.



(d) Variation on the number of functions per language/language.



(e) Variation on the size of executable per language/language.

Figure 4: Barplots to illustrate the variation of shellcode samples.

Language	Compiler/Packager	VT1	Detection Sig 1	Capa Detection 1	VT2	Detection Sig 2	Capa Detection 2
Ada	GNAT	1/70	fp	✗	23/73	✓	✓
Assembly	YASM/Golink	9/68	✓	✓	29/68	✓	✓
AutoHotKey	Ahk2EXE	9/68	✓	✓	5/72	✓	✗
AutoIt	Au2EXE	12/70	✓	✓	32/69	✓	✓
C	DMC	4/69	✓	✓	22/71	✓	✓
C	TinyC	5/70	✓	✓	45/72	✓	✓
C	BCC	6/68	✓	✓	21/70	✓	✓
C	mingw/gcc	22/72	✓	✓	51/73	✓	✓
C	msvc/cl	17/73	✓	✓	37/73	✓	✓
C#	bflat	7/71	✓	✓	1/70	fp	✗
C#	msc	0/69	✗	✓	21/73	✓	✓
C#	csc	1/73	fp	✓	5/73	✓	✓
C++	cl	17/70	fp	✗	34/73	✓	✓
C++	icl	17/70	fp	✗	17/73	✓	✓
C++	g++	5/73	✓	✓	36/73	✓	✓
Clojure	graal-vm	0/73	✗	✗	15/73	✓	✗
CommonLisp	sbcl	0/72	✗	✗	0/72	✗	✗
Crystal	crystal	3/73	fp	✗	15/73	✓	✓
D	dmd	5/66	fp	✗	6/73	✓	✗
Dart	dart	0/70	✗	✗	5/69	✓	✗
Eiffel	ec	0/67	✗	✗	11/68	✓	✓
F#	fsharpc	3/71	fp	✓	22/72	✓	✓
Fortran	ifort	3/76	fp	✗	17/72	✓	✓
GnuCobol	cobc	4/72	✓	✓	23/73	✓	✓
Golang	go	4/70	✓	✗	16/69	✓	✗
Groovy	Launch4j	2/66	fp	✓	4/62	✓	✗
Haskell	GHC	0/71	✗	✗	1/66	fp	✗
IronPython	ipyc	2/67	fp	✓	2/67	fp	✗
Java	graal-vm	1/73	fp	✗	2/73	fp	✗
Javascript	deno	0/65	✗	✗	0/68	✗	✗
Jscript	jsc	2/67	fp	✗	16/73	✓	✓
Kotlin	graal-vm	2/63	fp	✗	1/73	fp	✗
Kotlin	kotlin-native	0/68	✗	✗	1/67	fp	✗
Lua	luastatic	1/69	fp	✓	14/72	✓	✗
Nim	nim	0/70	✗	✗	25/69	✓	✗
ObjectiveC	gcc	2/68	fp	✓	25/69	✓	✗
Pascal	fpc	0/66	✗	✓	11/66	✓	✓
Perl	par	3/70	fp	✗	1/71	fp	✗
Phix	phix	10/72	✓	✗	21/67	✓	✗
PureBasic	pbcompiler	1/68	fp	✓	23/67	✓	✓
Python	pyinstaller	6/67	✓	✗	3/68	fp	✗
Python	nuitka	0/69	✓	✗	5/71	✓	✗
Racket	raco	0/64	✗	✗	1/64	fp	✗
Red	red	16/69	✓	✓	22/66	✓	✓
Ruby	ocra/aibica	26/68	✓	✗	2/71	fp	✗
Rust	rustc	0/71	✗	✗	16/72	✓	✗
Scala	graal-vm	0/73	fp	✗	1/73	fp	✗
Scala	launch4j	4/67	fp	✗	5/63	✓	✗
VB .NET	vbc	5/69	✓	✓	13/70	✓	✓
Zig	zig	0/73	✗	✗	19/68	✓	✓

Table 1: VirusTotal and Capa results for various programming languages and compilers/packagers.

`push <byte>`). Since the objective of this experiment was to statically locate the raw dummy payload using static methods and not actually locate the shellcode by any means necessary, we did not use any dynamic analysis tools.

After executing the script, all identified patterns were manually reviewed using a debugger and a hex editor to confirm the matches and eliminate false positives. This step ensured that only genuine shellcode patterns were considered in the results. Table 2 categorizes the matches into four levels of fragmentation, namely: (1) **None**: Shellcode bytes were sequential, indicating that there was no fragmentation; (2) **Medium**: Shellcode bytes were scattered but with gaps within a range; (3) **Heavy**: Shellcode bytes were fragmented with scattered chunks of large distance, wherein each chunk bytes was sequential or had fixed gaps within a range; (4) **N/A**: The script was unable to confidently identify the shellcode in the binary, indicating the highest level of fragmentation or potential complex encoding.

The results showed considerable discrepancies in pattern matching; for example, samples written in languages such as C and C++ retained, usually all shellcode bytes in sequential order or had a fixed gap between the bytes, leading to relatively straightforward detection. However, other languages demonstrated significant byte fragmentation and variations in memory layout, complicating static detection. For instance, our Rust implementation showed a complex pattern with the shellcode bytes dispersed irregularly throughout the binary at various offsets (e.g., starting with an initial block of 192 bytes at `0x16570` with no gaps, followed by a smaller, non-sequential block with gaps of up to 13 bytes at `0x4ee` and continued again with non-continuous blocks of shellcode at address `0x16630`). In the unique case of Phix, the shellcode was pushed on the stack byte by byte. Finally, in languages like Lisp and Haskell, we could not find any part of the shellcode with confidence. Another interesting result is that most of the samples with low detection rates also had their shellcode pattern unidentified within the binary, indicating another correlation between their structure and evasive behavior.

4.5 Reverse Engineering Metrics

In this section, we try to measure the shellcode binaries from the runtime complexity perspective. Although reverse engineering difficulty is not easy to measure as it is heavily based on the human element of intuition and expertise as well as on how fine-tuned

Language	Compiler/Packager	Fragmentation	Section Stored	Matched Ratio
Ada	GNAT	none	.rdata	1
Assembly	YASM/Golink	none	.data	1
AutoHotKey	Ahk2EXE	N/A	N/A	N/A
AutoIt	Au2EXE	N/A	N/A	N/A
C	DMC	none	CRT$XIA	1
C	TinyC	medium	.text	1
C	BCC	none	.data	1
C	mingw/gcc	none	.rdata	1
C	msvc/cl	none	.data	1
C#	bflat	none	.rdata	1
C#	msc	none	.sdata	1
C#	csc	none	.text	1
C++	cl	medium	.text	1
C++	icl	none	.rdata	1
C++	g++	none	.rdata	1
Clojure	graal-vm	none	.svm_hea	1
CommonLisp	sbcl	N/A	N/A	N/A
Crystal	crystal	heavy	.rdata	0.86
D	dmd	heavy	.text	0.93
Dart	dart	heavy	.text	0.62
Eiffel	ec	medium	.text	1
F#	fsharpc	heavy	.text	0.31
Fortran	ifort	none	.data	1.0
GnuCobol	cobc	none	.rdata	1.0
Golang	go	none	.rdata	1.0
Groovy	Launch4j	N/A	N/A	N/A
Haskell	GHC	N/A	N/A	N/A
IronPython	ipyc	medium	.text	1
Java	graal-vm	medium	.text	1
Javascript	deno	N/A	N/A	N/A
Jscript	jsc	medium	.text	1
Kotlin	graal-vm	medum	.text	1
Kotlin	kotlin-native	medium	.text	1
Lua	luastatic	N/A	N/A	N/A
Nim	nim	none	.data	1
ObjectiveC	gcc	none	.text	1
Pascal	fpc	medium	.text	1
Perl	par	N/A	N/A	N/A
Phix	phix	medium	.text	1
PureBasic	pbcompiler	none	.data	1
Python	pyinstaller	N/A	N/A	N/A
Python	nuitka	N/A	N/A	N/A
Racket	raco	N/A	N/A	N/A
Red	red	none	.data	1
Ruby	ocra/aibica	N/A	N/A	N/A
Rust	rustc	heavy	.rdata/.text	1
Scala	graal-vm	medium	.text	1
Scala	launch4j	N/A	N/A	N/A
VB .NET	vbc	medium	.text	1
Zig	zig	none	.text	1

Table 2: Shellcode fragmentation through pattern matching on binaries.

the used tools are, our high-level metrics indicate a connection between our most evasive samples and their actual complexity. In particular, we focused on the following key metrics (see Tables 3 and 4): number of functions, number of functions actually executed, average function size of executed functions, unique basic blocks executed, unique instructions executed based on the address they were found, meaning that if a particular instruction of a basic block is traversed more than once, it is not counted.

We also calculated the average cyclomatic complexity of the executed functions, the unique indirection calls and jumps executed, as well as the number of threads spawned. To acquire our results, we collected complete instruction traces of the executables with the help of IDAPro and its PinTracer debugger without taking into account traces from Windows dynamic libraries. We intentionally focused on indirect jumps and calls while excluding other control flow operands (for example, returns). This decision was motivated by the desire to capture the control flow aspects that most significantly impact program complexity and dynamic behavior and create static analysis challenges. Indirect control flow transfers, such as indirect jumps and indirect calls, are crucial in representing dynamic behavior in programs. They occur when the target of a jump or call is determined at runtime, often through function pointers, virtual method tables, or dynamic dispatch mechanisms.

By concentrating on indirect calls and jumps, we essentially measure how much of the program's control flow is determined at runtime rather than at static analysis time. A high number or a dense network of indirect branches can suggest more dynamic behavior, code unintended obfuscation techniques, or pointer-based dispatch tables, all of which add to reverse engineering difficulty. To that end, we also constructed CFGs that capture the indirection aspect of the shellcode samples where each node represented an indirect `jmp` or `call` and measure the total traversals, which indicate how many indirections occurred in total during execution. A large number of edge traversals can imply that the program frequently relies on indirect control flow to reach various parts of the code, suggesting that any attempt at reverse engineering must continuously resolve these runtime-dependent branches. We further incorporate an information-theoretic measure; Shannon's entropy to capture the unpredictability of edge transitions.

Entropy, computed from edge frequencies, quantifies how the transitions are distributed among possible indirect edges. High entropy indicates that the program does not favor a small set of indirect branches but rather exercises many of them with similar frequency, increasing uncertainty for the analyst. On the contrary, a low-entropy graph may be complex in structure but predictable in practice if only a few edges are predominantly taken.

We also need to mention that the sizes of the trace log files ranged from a few kilobytes to almost 10 Gigabytes. We also did not include in the analysis the .NET languages except for the C#-csc sample, which was the only sample that included the runtime environment, since usually, .NET compiled languages do not need to include the runtime environment to be able to run in a target. Also, these samples can be trivially reversed using dnSpy and .NET-focused tools.

We observed that in almost all cases that reported low detection scores, there were many indirections or large and complex functions, showcasing how the runtime environment of each language adds vast amounts of complexity to simple malicious code.

4.6 Case Study: Haskell Reverse Engineering

In this section, we chose to investigate the challenges posed by reverse engineering one of our shellcode samples that had evasive behavior, in particular the Haskell executable, focussing on how the inherent characteristics of the GHC runtime and its execution model complicate traditional analysis techniques compared to the corresponding C sample compiled with the Windows MSVC toolchain. Our goal is to highlight some of the key differences we observed that introduce substantial complexity and how static disassembly or debugging struggles to provide accurate insights into their execution. However, providing a complete analysis of the binary is beyond the scope of this work, as this would require a deep dive into the GHC runtime and Lambda calculus.

In C, control flow is generally direct and imperative. Each instruction or function call follows in a predictable sequence, and system calls, such as memory allocation through `VirtualAlloc`,

Language	Compiler	#Func	#Func Exec	Avg Func Size	#BB Hits	#Instr Hits	CC	#Ind Jmps	#Ind Calls
Ada	GNAT	1695	92	171.08	493	2482	3.51	8	36
Assembly	YASM/Golink	5	5	19	5	26	1	0	0
AutoHotKey	Ahk2EXE	1464	147	1169.82	3606	15128	48.44	23	12
AutoIt	Au2EXE	2282	132	287.77	378	8441	9.65	0	44
C	DMC	69	34	106.53	186	902	4.94	0	0
C	TinyC	15	10	215.3	10	500	1	0	0
C	BCC	309	65	101.14	69	783	3.16	0	1
C	mingw/gcc	79	13	98.24	18	482	4.03	0	5
C	msvc/cl	436	47	129.43	91	1061	4.66	2	0
C#	bflat	3718	349	166.68	683	9769	4.43	6	16
C#	csc	17736	784	142.76	440	4354	8052	36	0
C++	cl	343	26	141.3	6	392	3.81	0	0
C++	icl	451	37	161.54	74	993	5.17	0	0
C++	g++	79	33	98.24	93	445	4.03	0	5
Clojure	graal-vm	7314	1042	1284.87	13436	133483	31.32	7	564
CommonLisp	sbcl	781	195	560	2087	26931	134.4	1	101
Crystal	crystal	3327	193	203.16	586	5682	6.98	4	6
D	dmd	2409	1429	164.5	720	10982	4.13	5	32
Dart	dart	9251	916	308.88	2167	40830	6.86	13	141
Eiffel	ec	4051	762	146.58	894	18068	2.97	0	4
Fortran	ifort	914	291	492.85	2183	11009	17.75	21	1
GnuCobol	cobc	100	22	95.8	45	227	2.90	0	0
Golang	go	1616	439	382.97	4478	35007	1.77	2	21
Groovy	Launch4j	162	130	131.31	364	4068	4.92	0	1
Haskell	GHC	2974	2318	187.3	2200	22596	4.97	276	47
Java	graal-vm	6969	996	969.05	12764	125244	23.35	6	413
Javascript	deno	81792	1717	460.99	37475	280860	9.75	1521	0
Kotlin	graal-vm	6902	981	973.44	12955	55424	23.4	5	431
Kotlin	kotlin-native	1574	206	150.85	574	10582	4.67	3	26
Lua	luastatic	1545	332	350.55	2821	16246	10.16	54	29
Nim	nim	359	130	226.28	309	5343	2.26	0	24
ObjectiveC	gcc	52	24	113.2	43	291	2.27	0	0
Pascal	fpc	429	145	128.86	305	4051	3.77	0	41
Perl	par	2821	82	146.79	276	15570	4.71	5	431
Phix	phix	167	82	522.39	390	1842	22.46	0	11
PureBasic	pbcompiler	44	10	36.30	2	113	1.10	1	0
Python	pyinstaller	819	117	302.7	577	6075	10.77	4	22
Python	nuitka	370	79	670.63	1234	5841	12.92	3	19
Racket	raco	116	49	148.71	328	2219	4.51	0	49
Red	red	22	8	99.0	13	224	1.25	0	0
Ruby	ocra/aibica	132	63	234.63	488	3077	5.98	0	48
Rust	rustc	337	36	103.5	95	595	2.42	2	4
Scala	graal-vm	7021	967	1019.57	13186	130330	23.61	5	433
Scala	launch4j	167	116	142.51	432	4050	4.79	0	1
Zig	zig	639	212	374.8	1191	10269	2.05	4	11

Table 3: Reverse engineering metrics I.

```
lea     rax, [rsp+188h+var_148]
lea     rcx, unk_7FF7EDACB000
mov     rdi, rax
mov     rsi, rcx
mov     ecx, 115h
rep movsb
mov     r9d, 40h ; '@'  ; flProtect
mov     r8d, 3000h      ; flAllocationType
mov     edx, 115h       ; dwSize
xor     ecx, ecx        ; IP address
call    cs:VirtualAlloc
mov     [rsp+188h+lpStartAddress], rax
mov     r8d, 115h
lea     rdx, [rsp+188h+var_148]
mov     rcx, [rsp+188h+lpStartAddress]
call    sub_7FF7EDABFBA0
```

Listing 3: Disassembled snippet of C shellcode sample.

are explicit and immediately visible. For instance, in our sample, the first thing an analyst would see in the disassembled code is that the steps are linear (see Listing 3).

Furthermore, since the disassembled code is straightforward, it can be analyzed statically in a few minutes. In (Listing 3), the shellcode is loaded from the section in which it resides (.data) in `rcx`, and then it gets copied in the stack space byte by byte. Then, `VirtualAlloc` allocates some space, and the actual shellcode gets copied from the stack to the fresh `RWX` allocated space.

On the other hand, reversing Haskell binaries presents significant challenges due to the intricacies of its execution model. Going from the call to `VirtualAlloc` to copying shellcode into the allocated space involves more than 100 thousand instructions based on the execution trace we got as opposed to the C sample, which was only five instructions. Throughout the disassembled code, user code is blended with the STG-machine code that handles each own stack and heap, has sophisticated pointer management and garbage collector, and also makes heavy use of indirection jumps because of its lazy evaluation that defers computation until needed and continuation-passing.

A continuation is a callback function that expects the result of a previous computation as an argument; in other words, it represents 'what to do next.' Closures and continuations are among the main reasons Haskell reported such a high number of indirections. Here, the shellcode is initialized from the raw binary written in heap memory, then prepared for the interaction with the FFI, and inherits a unique obfuscation scheme to execute the shellcode. The code in Listing 4 shows how each byte of the shellcode is stored in the raw executable and what is executed during the initialization of our malicious shellcode in heap memory. Considering that `rbp` and `r12` are the equivalent of stack and heap registers in STG-machine, the code after a series of memory checks goes through a memory allocation routine using `newCAF` and `allocateMightFail`

Language	#Nodes	#Edges	#Traversals	#Tot. Ind Cals	#Tot. Ind Jmps	CFG Entropy
Ada	44	45	74	63	12	0.98
Assembly	0	0	0	0	0	0
AutohotKey	35	64	4973	1403	3571	0.66
AutoIt	44	73	5983	1678	4305	0.57
C-bcc	1	1	21	0	52	0
C-cl	2	2	3	0	4	0.91
C-gcc	5	4	4	5	0	1.0
C-tcc	0	0	0	0	0	0
C-dmc	0	0	0	0	0	0
C#-bflat	22	34	24559	329	24321	0.53
C#-csc	36	127	237	0	237	0.43
C++-cl	0	0	0	0	0	0
C++-icl	0	0	0	0	0	0
C++-g++	5	4	4	5	0	1.0
Clojure	571	890	19176	18853	324	0.53
CommonLisp	102	126	706	693	14	0.54
Crystal	10	18	2088	1032	1057	0.30
D	37	53	199	186	14	0.58
Dart	154	249	34673	14750	19924	0.41
Eiffel	4	7	41	42	0	0.74
Fortran	22	26	55	1	55	0.93
GnuCobol	0	0	0	0	0	0
Golang	23	69	6219	6057	163	0.34
Groovy	1	0	0	1	0	0
Haskell	323	652	8265	488	7778	0.66
Java	419	634	19879	19732	263	0.57
Javascript	1521	3427	403815	403815	0	0.56
Kotlin-graalvm	436	660	19917	19657	261	0.56
Kotlin-native	29	41	189	187	3	0.63
Lua	83	220	3753	869	2885	0.57
Nim	24	30	35	36	0	0.98
ObjC	0	0	0	0	0	0
Pascal	41	56	88	89	0	0.96
Perl	436	660	19917	19657	261	0.56
Phix	11	25	30967	30968	11	0.24
PureBasic	1	0	0	0	1	0
Python-pyinstaller	26	37	563	453	110	0.51
Python-nuitka	22	27	76	38	39	0.89
Racket	49	70	795	796	0	0.62
Red	0	0	0	0	0	0
Ruby	48	89	432	432	0	0.65
Rust	6	6	6	5	2	1.0
Scala-graalvm	438	669	20207	19945	263	0.55
Scala-launch4j	1	0	0	1	0	0
Zig	15	24	171	17	155	0.50

Table 4: Reverse engineering metrics II.

```
lea     rax, [rbp-20h]
cmp     rax, r15
jb      short loc_40BE86
add     r12, 10h
cmp     r12, [r13+358h]
ja      short loc_40BE7B
sub     rsp, 8
mov     rcx, r13
mov     rdx, rbx
sub     rsp, 20h
xor     eax, eax
call    newCAF
add     rsp, 28h
test    rax, rax
jz      short loc_40BE79
mov     qword ptr [rbp-10h], 43BFF8h
mov     [rbp-8], rax
mov     qword ptr [r12-8], 4C01C8h
mov     qword ptr [r12], 0FCh
lea     rax, [r12-7]
mov     r14d, offset base_GHCziWord_zdfNumWord8_closure
mov     qword ptr [rbp-20h], 43D1A0h
mov     [rbp-18h], rax
add     rbp, 0FFFFFFFFFFFFFFE0h
jmp     base_GHCziNum_fromInteger_info
```

Listing 4: Disassembled snippet of Haskell shellcode sample.

GHC functions. Finally, the instruction `mov qword ptr [r12], 0FCh` stores the first byte of our shellcode (`0xFC`) at the address pointed to by `r12` in heap memory. During what we just described, many other procedures occur, such as thread handling and garbage collection, making the code even more incomprehensible than the Assembly produced by C.

As we saw, the executable inherits from the actual language runtime an obfuscation scheme where the shellcode is stored and loaded dynamically byte by byte and is only fully assembled in executable memory at runtime.

5 Discussion

Languages such as Java, Clojure, Scala, Kotlin, and JavaScript, which embed substantial runtimes or rely on JIT compilation, consistently produced large, complex binaries. These executables exhibited extensive CFGs (high node/edge counts), numerous indirect calls/jumps, and large numbers of functions. VirusTotal results showed that such complexity often correlated with higher detection rates or initial false positives. Heuristic-based detection engines frequently flagged these binaries as suspicious, likely due to unfamiliar or intricate patterns in control flow and the presence of runtime scaffolding code. Although subsequent passes or capa reports sometimes clarified these detections, the initial suspicion underscores the challenges static AV tools face when analyzing runtime-heavy executables.

In contrast, binaries produced by traditional compiled languages (C, Fortran, Ada) and straightforward compilers tended to have simpler structures. With fewer functions, less fragmentation, and minimal indirect control flows, these binaries were more transparently analyzable. Their matched ratios were often perfect (1.0), indicating easy alignment between the binary and static analysis tools. As a result, detection outcomes were more predictable. Such binaries were either not detected at all or consistently identified as benign. When detections occurred, they were more easily interpreted, reducing the likelihood of persistent false positives.

Heavy fragmentation corresponded to lower matched ratios, complicating static analysis and potentially increasing false-positive rates. Fragmented code segments impeded effective disassembly and structured understanding of the binary. As a result, AV engines that rely on pattern matching or heuristic scanning may misinterpret such binaries as suspicious, even without known malicious signatures.

The prominence of indirect calls and jumps in runtime-heavy languages serves as an additional complexity signal. Indirect branching complicates the control-flow analysis, challenging both AV signatures and CFG extraction tools. The correlation between indirect control-flow patterns and AV detections or FPs suggests that complexity in flow redirection can raise the heuristic suspicion threshold and lead to detections.

Finally, normalized entropy provided insights into the uniformity of byte distributions. High entropy often occurs in packed or obfuscated binaries, which can appear anomalous to AV engines as most modern malware uses some packer. While not the sole predictor of detection outcomes, elevated entropy combined with fragmentation and complex control-flow patterns often coincided with uncertain or cautious AV responses.

Our results highlight that no single metric conclusively determines AV detection outcomes. Instead, a combination of factors—runtime complexity, fragmentation, control-flow intricacy, entropy, and function-level distributions—influences how AV engines classify binaries. Therefore, from the defender's and analysts' perspectives, understanding these correlations, creating more robust signatures, and extending the scope of tools to consider more programming languages and compilers is imperative, as threat actors can easily exploit this gap. From an attacker's perspective,

the findings indicate that complexity and indirect control flow can serve as evasive techniques, potentially raising false alarms or complicating detection. However, sustained complexity may also attract scrutiny, highlighting a delicate balance between obfuscation and detection risk.

6 Conclusion

Malware is predominantly written in C/C++ and is compiled with Microsoft's compiler. However, trying to answer **RQ1** with our experiments, our work practically shows that by shifting the codebase to another, less used programming language or compiler, malware authors can *significantly decrease the detection rate* of their binaries but simultaneously *increase the reverse engineering effort* of the malware analysts. It is crucial to note that the malware authors do not necessarily need to radically change their codebase, as, for instance, the choice of another compiler, even for famous programming languages like C, can have the same impact. Our experimental results illustrate that there are significant deviations in how programming languages and compilers generate binaries, and that they can serve as an additional layer of obfuscation for malware authors.

The root cause for the disparities that we raise (**RQ2**), as highlighted with our use case in Haskell and the metrics for each tested pair of programming language and compiler, is that there are radically different ways that each of them reaches the same result. For instance, different ways of storing strings and different approaches in the internal representation of functions can render many static detection rules useless. As a result, there is no "one-size-fits-all" approach, so further research is necessary to systematically identify these differences and group them.

Moreover, answering **RQ3**, this shift may come with additional benefits for attackers. An obvious case is cross-compilation and multi-platform targeting languages, which enable malware authors to build a single malware variant and have it compiled for multiple operating systems. This strategy can significantly reduce the time and number of tools needed to achieve their objectives, thereby expanding the scope of any hostile campaign. IoT devices, in particular, support a range of CPU environments, making it necessary for malware targeting these devices to be compatible with not only x86 and x64 architectures but also various other architectures such as ARM, MIPS, m68k, SPARC, and SH4.

A typical example is Mirai [5], which uses GCC, yet one of its successors, NoaBot [4], uses uClibc-based cross-compiler and is statically built to target embedded Linux systems. In this regard, other options could be more efficient. For instance, Go can be cross-compiled to all major operating systems, as well as Android, JavaScript, and WebAssembly. One of its advantages is that it provides statically compiled binaries by default, eliminating runtime dependencies and simplifying deployment on target systems. Go also features a robust package ecosystem that allows developers to easily pull in code from other sources. In general, cross-compilation in Go is as simple as setting two environment variables, making it almost trivial to modify the build process to produce binaries for every major platform. As a result, malware can be developed at a faster rate, targeting a broader range of architectures and systems. Indeed, HinataBot [3], a descendant of Mirai, is developed in Go to take advantage of the above. The HinataBot was more difficult to be discovered by detection systems. Unfortunately, the bar to creating a new variant of Mirai using Go or other languages is low, and criminal groups make their own variations [2].

Beyond cross-compilation, there are several other reasons to witness more changes in the malware codebase. After all, malware developers, like any other software engineers, have specific needs when choosing programming languages and tools. Different languages offer various benefits for different scenarios, and the choice of language can significantly impact the development and functionality of malware. For instance, built-in security mechanisms and type safety may be prioritized by ransomware authors who want to avoid leaks of the encryption keys to guarantee that their victims will not be able to develop decryptors. A typical example is Rust, which offers built-in memory mechanisms to prevent common vulnerabilities and type safety. Other aspects can include library availability; facilitating interaction with the underlying operating system and enabling critical malware functions, low-level access, and control over memory layout; having full control over the malware's behavior and performance but also direct compilation to machine code; creating an executable file directly and use other tools for obfuscation.

While shifting to another programming language may seem complicated, especially when considering less popular ones, large language models (LLMs) may come to the rescue; after all, they have proven their capacity in generating code quite accurately [16, 22, 23, 35, 41] and various cybersecurity tasks [12, 37], and malicious actors are abusing them. As a result, they can translate code from one programming language to another, requiring little fine-tuning. This way, malware authors can seamlessly develop loaders, droppers, and other components in languages they may not be familiar with.

It is true that the malware that we examine in this work represents a small fragment of the total; nevertheless, it is stealthier and introduces more bottlenecks for the reverse engineer. Given that the APT groups are shifting their codebases and the malware-as-a-service model facilitates the trading of malware so different malware mixtures per campaign can be purchased, this diversification is expected to continue. By disregarding these samples and only focusing on traditional programming languages and compilers, we provide malware authors with an effective hideout that they can easily exploit. Therefore, we believe that a deeper analysis of the executables produced by other compilers and programming languages is needed to improve detection rates but also develop better reverse engineering tools.

Acknowledgments

This work was supported by the European Commission under the Horizon Europe Programme as part of the project SafeHorizon (Grant Agreement no. 101168562). The content of this article does not reflect the official opinion of the European Union. The responsibility for the information and views expressed therein lies entirely with the authors.

References

[1] Abuse.ch. 2024. Malware Bazaar. https://bazaar.abuse.ch/.
[2] Antonia Affinito, Stefania Zinno, Giovanni Stanco, Alessio Botta, and Giorgio Ventre. 2023. The evolution of Mirai botnet scans over a six-year period. *Journal of Information Security and Applications* 79 (2023), 103629.

[3] Akamai. 2024. Uncovering HinataBot: A Deep Dive into a Go-Based Threat. https://www.akamai.com/blog/security-research/hinatabot-uncovering-new-golang-ddos-botnet.

[4] Akamai. 2024. You Had Me at Hi — Mirai-Based NoaBot Makes an Appearance. https://www.akamai.com/blog/security-research/mirai-based-noabot-crypto-mining.

[5] Manos Antonakakis, Tim April, Michael D. Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. 2017. Understanding the Mirai Botnet. In *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, Engin Kirda and Thomas Ristenpart (Eds.). USENIX Association, 1093–1110.

[6] Theodoros Apostolopoulos. 2025. https://github.com/nihilboy/Coding-Malware-in-Fancy-Programming-Languages-for-Fun-and-Profit.

[7] AV-ATLAS. 2024. https://portal.av-atlas.org/malware.

[8] Rosangela Casolare, Giovanni Lacava, Fabio Martinelli, Francesco Mercaldo, Marco Russodivito, and Antonella Santone. 2022. 2Faces: a new model of malware based on dynamic compiling and reflection. *Journal of Computer Virology and Hacking Techniques* 18, 3 (Sept. 2022), 215–230. doi:10.1007/s11416-021-00409-8

[9] CERT-UA. 2023. APT28: from initial attack to creating threats to a domain controller in an hour (CERT-UA#8399). https://cert.gov.ua/article/6276894. (In Ukranian).

[10] Eric Chien, Liam OMurchu, and Nicolas Falliere. 2012. W32.Duqu: The Precursor to the Next Stuxnet . In *5th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET 12)*.

[11] Pierre Delcher. 2023. Tomiris called, they want their Turla malware back. https://securelist.com/tomiris-called-they-want-their-turla-malware-back/109552/#jlorat.

[12] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, et al. 2023. PentestGPT: An LLM-empowered Automatic Penetration Testing Tool. *arXiv preprint arXiv:2308.06782* (2023).

[13] ENISA. 2021. ENISA Threat Landscape 2021. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2021/.

[14] European Union Agency for Cybersecurity (ENISA). 2023. Threat Landscape Report 2023. https://www.enisa.europa.eu/publications/enisa-threat-landscape-2023.

[15] Lorena González-Manzano, José M de Fuentes, Flavio Lombardi, and Cristina Ramos. 2023. A technical characterization of APTs by leveraging public resources. *International Journal of Information Security* 22, 6 (2023), 1567–1584.

[16] Yuejun Guo, Constantinos Patsakis, Qiang Hu, Qiang Tang, and Fran Casino. 2024. Outside the Comfort Zone: Analysing LLM Capabilities in Software Vulnerability Detection. In *Computer Security - ESORICS 2024 - 29th European Symposium on Research in Computer Security*.

[17] George Karantzas and Constantinos Patsakis. 2021. An Empirical Assessment of Endpoint Detection and Response Systems against Advanced Persistent Threats Attack Vectors. *Journal of Cybersecurity and Privacy* 1, 3 (2021), 387–421.

[18] Vasilios Koutsokostas and Constantinos Patsakis. 2021. Python and Malware: Developing Stealth and Evasive Malware Without Obfuscation. doi:10.48550/arXiv.2105.00565 arXiv:2105.00565 [cs].

[19] Igor Kuznetsov. 2012. The mystery of Duqu Framework solved. https://securelist.com/the-mystery-of-duqu-framework-solved-7/32354/.

[20] Ravie Lakshmanan. 2021. A Rust-based Buer Malware Variant Has Been Spotted in the Wild. https://thehackernews.com/2021/05/a-new-buer-malware-variant-has-been.html.

[21] Pengwei Lan, Pei Wang, Shuai Wang, and Dinghao Wu. 2017. Lambda Obfuscation. In *Security and Privacy in Communication Networks - 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22-25, 2017, Proceedings (Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Vol. 238)*. Springer, 206–224.

[22] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by ChatGPT really correct? Rigorous evaluation of Large Language Models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

[23] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE Transactions on Software Engineering* 50, 6 (2024), 1548–1584. doi:10.1109/TSE.2024.3392499

[24] Kenny Zhuo Ming Lu. 2019. Control flow obfuscation via CPS transformation. In *Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (POPL '19)*. ACM, 54–60. doi:10.1145/3294032.3294083

[25] Mandiant. 2024. capa. https://github.com/mandiant/capa.

[26] Marion Marschalek. 2014. Not old enough to be forgotten: the new chic of Visual Basic 6. https://www.virusbulletin.com/virusbulletin/2014/07/not-old-enough-be-forgotten-new-chic-visual-basic-6.

[27] Oscar Minks. 2021. The REBOL Yell: A New Novel REBOL Exploit. https://frsecure.com/blog/the-rebol-yell-new-rebol-exploit/.

[28] MITRE ATT&CK. 2024. APT29 Group. https://attack.mitre.org/groups/G0016/.

[29] MITRE Engenuity Center for Threat-Informed Defense. 2025. Sightings Ecosystem v2.0.0. https://center-for-threat-informed-defense.github.io/sightings_ecosystem/key-results/.

[30] MITTRE ATT&ACK. 2024. Command and Scripting Interpreter. https://attack.mitre.org/techniques/T1059/.

[31] MITTRE ATT&ACK. 2024. Process Injection. https://attack.mitre.org/techniques/T1055/.

[32] Nathaniel Mott. 2022. Hacking in tongues: Malware authors shake up their programming languages. https://readme.synack.com/hacking-in-tongues-malware-authors-shake-up-their-programming-languages.

[33] Microsoft Threat Intelligence Center (MSTIC). 2022. Hive ransomware gets upgrades in Rust. https://www.microsoft.com/en-us/security/blog/2022/07/05/hive-ransomware-gets-upgrades-in-rust/.

[34] James Nutland and Michael Szeliga. 2024. Akira ransomware continues to evolve. https://blog.talosintelligence.com/akira-ransomware-continues-to-evolve/.

[35] Shuyin Ouyang, Jie M Zhang, Mark Harman, and Meng Wang. 2023. LLM is Like a Box of Chocolates: the Non-determinism of ChatGPT in Code Generation. *arXiv preprint arXiv:2308.02828* (2023).

[36] Constantinos Patsakis, David Arroyo, and Fran Casino. 2024. The Malware as a Service ecosystem. In *Malware: Handbook of Prevention and Detection*. Springer, 371–394.

[37] Constantinos Patsakis, Fran Casino, and Nikolaos Lykousas. 2024. Assessing LLMs in malicious code deobfuscation of real-world malware campaigns. *Expert Syst. Appl.* 256 (2024), 124912. doi:10.1016/J.ESWA.2024.124912

[38] Andre Pawlowski, Moritz Contag, and Thorsten Holz. 2016. Probfuscation: An Obfuscation Approach using Probabilistic Control Flows. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer, 165–185.

[39] Meghna Praveen and Wesam Almobaideen. 2023. The Current State of Research on Malware Written in the Rust Programming Language. In *2023 International Conference on Information Technology (ICIT)*. 266–270. doi:10.1109/ICIT58056.2023.10226157

[40] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *2022 IEEE Symposium on Security and Privacy (SP)*. 1574–1589. doi:10.1109/SP46214.2022.9833626

[41] Fardin Ahsan Sakib, Saadat Hasan Khan, and A. H. M. Rezaul Karim. 2024. Extending the Frontier of ChatGPT: Code Generation and Debugging. In *2024 International Conference on Electrical, Computer and Energy Technologies (ICECET*. 1–6. doi:10.1109/ICECET61485.2024.10698405

[42] SECURELIST by Kaspersky. 2019. Zebrocy's Multilanguage Malware Salad. https://securelist.com/zebrocys-multilanguage-malware-salad/90680/.

[43] SentinelOne. 2024. SentinelOne and Intezer Team to Simplify Reverse Engineering of Rust Malware. https://www.sentinelone.com/press/sentinelone-and-intezer-team-to-simplify-reverse-engineering-of-rust-malware/.

[44] sKyWIper Analysis Team. 2012. sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks. https://ioactive.com/wp-content/uploads/2012/06/skywiper.pdf.

[45] TIOBE. 2024. TIOBE Index for December 2024. https://www.tiobe.com/tiobe-index/.

[46] US Cybersecurity and Infrastructure Security Agency (CISA). 2020. Cost of a Cyber Incident: Systematic Review and Cross-Validation.

[47] U.S. Cybersecurity and Infrastructure Security Agency (CISA). 2023. Cyber Incident Reporting for Critical Infrastructure Act of 2022 (CIRCIA). https://www.cisa.gov/circia.

[48] Javier Vicente and Brett Stone-Gross. 2022. Analysis of BlackByte Ransomware's Go-Based Variants. https://www.zscaler.com/blogs/security-research/analysis-blackbyte-ransomware-s-go-based-variants.

[49] VirusTotal. 2024. Virtus Total Search Engine and API. https://www.virustotal.com/.

[50] VirusTotal. 2024. VirusTotal Statistics. https://www.virustotal.com/gui/stats.

[51] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. 2016. Translingual Obfuscation. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. 128–144. doi:10.1109/EuroSP.2016.21

[52] Yan Wang, Shuai Wang, Pei Wang, and Dinghao Wu. 2018. Turing obfuscation. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*. Springer, 225–244.

[53] Miuyin Yong Wong, Matthew Landen, Frank Li, Fabian Monrose, and Mustaque Ahamad. 2024. Comparing Malware Evasion Theory with Practice: Results from Interviews with Expert Analysts. In *Twentieth Symposium on Usable Privacy and Security (SOUPS 2024)*. USENIX Association, Philadelphia, PA, 61–80.